Universidad
Andrés Bello

Programacion en C

Ingeniería en Computación e Informática

Universidad
Andrés Bello

Contenidos

Contenidos

El primer programa

Hola mundo

```
1  #include <stdio.h>
2  /*
3   Esto es una comentario
4   de mas lineas
5   */
6  int main() {
7      printf("Hola mundo\n"); // Esto es un comentario de una linea
8  }
```

Compilación:

```
$ gcc -o exe hola.c
$ ./exe
Hola mundo
```

Descripción del código

1 Se puede ver la inclusión de la librería `stdio`.

2-5 Es posible ver un comentario de multipes lineas, se indica colocando un `/*` al inicio y `*/` al final.

6 Se inicia el **bloque principal**, la forma de éste es la de una función.

7 Se imprime un mensaje a través de la salida estándar.

Variables y tipos de datos

Tipos de datos

```c
#include <stdio.h>

int main() {
    int a, b;
    float c, d;
    char e, f;

    a = 1;
    b = 100;
    c = 1.0;
    d = 0.8752;
    e = 'a';
    f = 'm';

    printf("% %\n", a, b);
    printf("% %\n", c, d);
    printf("% %\n", e, f);
}
```

Salida:

```
1 100
1.000000 0.875200
a m
```

Operadores aritméticos

```c
#include <stdio.h>

int main() {
    int a, b;
    float c, d, e;

    a = 10;
    b = 6;
    c = 0.5;
    d = 1.7;

    printf("%d + %f = %f\n", a, d, a + d);
    printf("%d - %f = %f\n", a, d, a - d);
    printf("%d*%f = %f\n", a, c, a*c);
    printf("%d/%f = %f\n", a, c, a/c);
    printf("%d mod %d = %d\n", a, b, a%b);
}
```

Salida:

```
10 + 1.700000 = 11.700000
10 - 1.700000 = 8.300000
10*0.500000 = 5.000000
10/0.500000 = 20.000000
10 mod 6 = 0.500000
```

Sentencias de control

Flujo de control

Las condiciones se componen de los **condicionales** y **operadores lógicos**, los que se pueden combinar.

```
1 if(condicion) {
2     //hacer cosas
3 }
```

```
1 if(condicion) {
2     // hacer cosas
3 } else {
4     // hacer estas cosas por defecto
5 }
```

```
1 if(condicion) {
2     // hacer cosas
3 } else if(otra condicion) {
4     // hacer cosas si ocurre esto
5 } else {
6     // hacer estas cosas por defecto
7 }
```

Sentencias de control

Condicionales y operadores lógicos

Cuadro: Condicionales

Operador	
$A == B$	Verdadero si A es igual a B
$A! = B$	Verdadero si A es distinto a B
$A < B$	Verdadero si A es menor a B
$A > B$	Verdadero si A es mayor a B
$A >= B$	Verdadero si A es mayor o igual a B
$A <= B$	Verdadero si A es menor o igual a B

Cuadro: Operadores lógicos

Operador	
$\|\|$	Operador OR
$\&\&$	Operador AND
!	Operador de negación

Ciclos

```
1  while(condicion) {
2      // cosas
3  }
```

Las sentencias **break** y **continue** permiten terminar el bucle completo (break) o continuar con la siguiente iteración (continue).

```
1  for(inicializacion; condicion; posterior) {
2      // cosas
3  }
4
5  /*********************************/
6  inicializacion
7  while(condicion) {
8      // cosas
9      posterior
10 }
```

Funciones

Una función es un conjunto de líneas de código que realizan una tarea específica y puede retornar un valor.

```
1 tipo_dato nombre_funcio(parametros) {
2     // cosas
3     return algo;
4 }
```

Ejemplo:

```
1 int paridad(int n) {
2     if (n%2 == 0) {
3         return algo;
4     } else {
5         return 0;
6     }
7 }
```

Librerías

```
1 #ifndef _ABB_H_
2 #define _ABB_H_
3
4 int primo(int n);
5
6 #endif
```

Código 1: mi_libreria.h

```
1  #include "mi_libreria.h"
2
3  int primo(int n) {
4      int a = 0, i;
5
6      for(i = 1; i <= n; i++) {
7          if(n% == 0) a++;
8      }
9
10     return a;
11 }
```

Código 2: mi_libreria.c

```
1  #include <stdio.h>
2  #include "mi_libreria.h"
3
4  int main(){
5      int a;
6      a=primo(13);
7
8      if(a==2) printf("El número es primo\n");
9      else printf("El número no es primo\n");
10
11     return 0;
12 }
```

Código 3: main.c

Librerías

Compilación

mi_libreria.h

mi_libreria.c

main.c

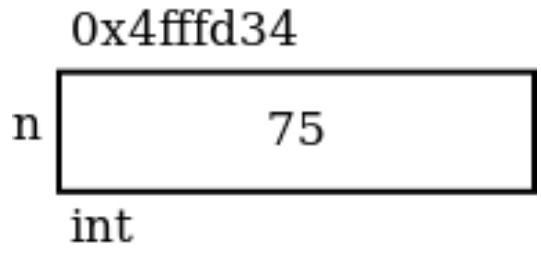
```
$gcc -o exe mi_libreria.c main.c
$ ./exe
Introduce un numero: 7
El número es primo
```

Punteros

Se asocian tres atributos fundamentales:

- nombre
- tipo
- dirección

```
1    int n;
```

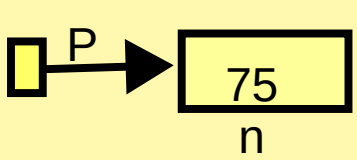



```
1  printf("%d\n", n);    // imprime el valor de n
2  printf("%d\n", &n);   // imprime la dirección de n
```

- Es una variable como cualquier otra.
- Contiene una dirección que apunta a otra posición de memoria.
- En esa posición se almacena los datos a los que apunta el puntero.
- Un puntero apunta a una variable de memoria.

Punteros

```
1  #include <stdio.h>
2
3  int main (int argc, char const* argv[])
4  {
5    int n = 75;
6    int  *p = &n; // p contiene la dirección de n
7
8    printf("n = %d, &n = %d\n", n, &n);
9    printf("p = %d, &p = %d\n", p, &p);
10
11   return 0;
12 }
```



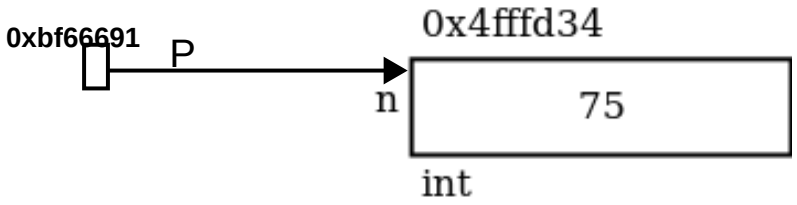
P apunta a un entero con identificador n

→ n no es puntero, sin embargo se puede mostrar su dirección de memoria con &n

PREGUNTA:¿CÓMO PUEDO IMPRIMIR EL VALOR DE n USANDO EL PUNTERO p?

Salida:

n = 75, &n = 0x4fffd34
p = 0x4fffd34, &p = 0xbf66691



Punteros

Ejemplo

```
1  #include <stdio.h>
2
3  int main (int argc, char const* argv[])
4  {
5    char c; // variable carácter
6    char *pc; // un puntero a una variable carácter
7    pc = &c;
8    for (c = 'A'; c <= 'Z'; c++)
9      printf("%c ", *pc);
10   printf("\n");
11
12   return 0;
13 }
```

primero se declara el puntero y luego en la línea 7 se inicializa, sin embargo, al igual en el ejemplo anterior, en el momento de declarar el puntero pc se puede inicializar así: char *pc = &c;

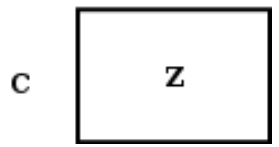
Salida:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Punteros

Puntero a puntero

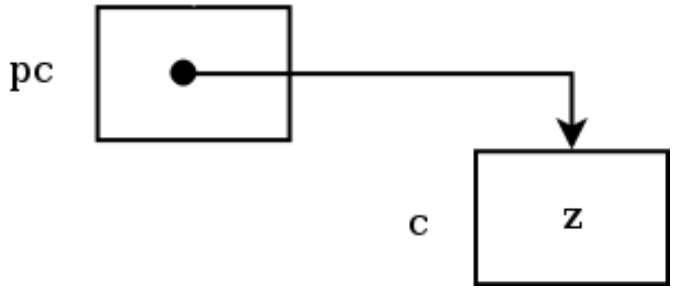
```
char c = 'z';
char *pc = &c;
char **ppc = &pc;
char *** pppc = &ppc;
***pppc = 'm'; // cambia el valor de c a 'm'
```



Punteros

Puntero a puntero

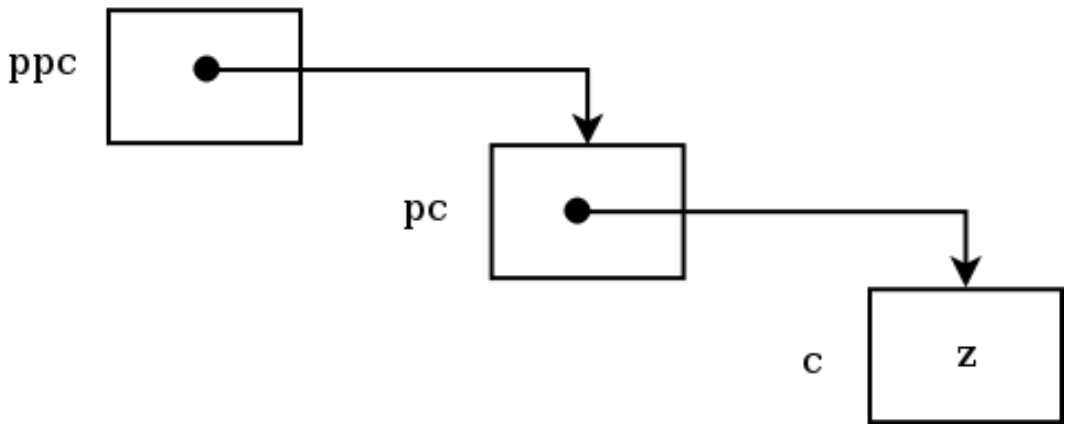
```
char c = 'z';
char *pc = &c;
char **ppc = &pc;
char *** pppc = &ppc;
***pppc = 'm'; // cambia el valor de c a 'm'
```



Punteros

Puntero a puntero

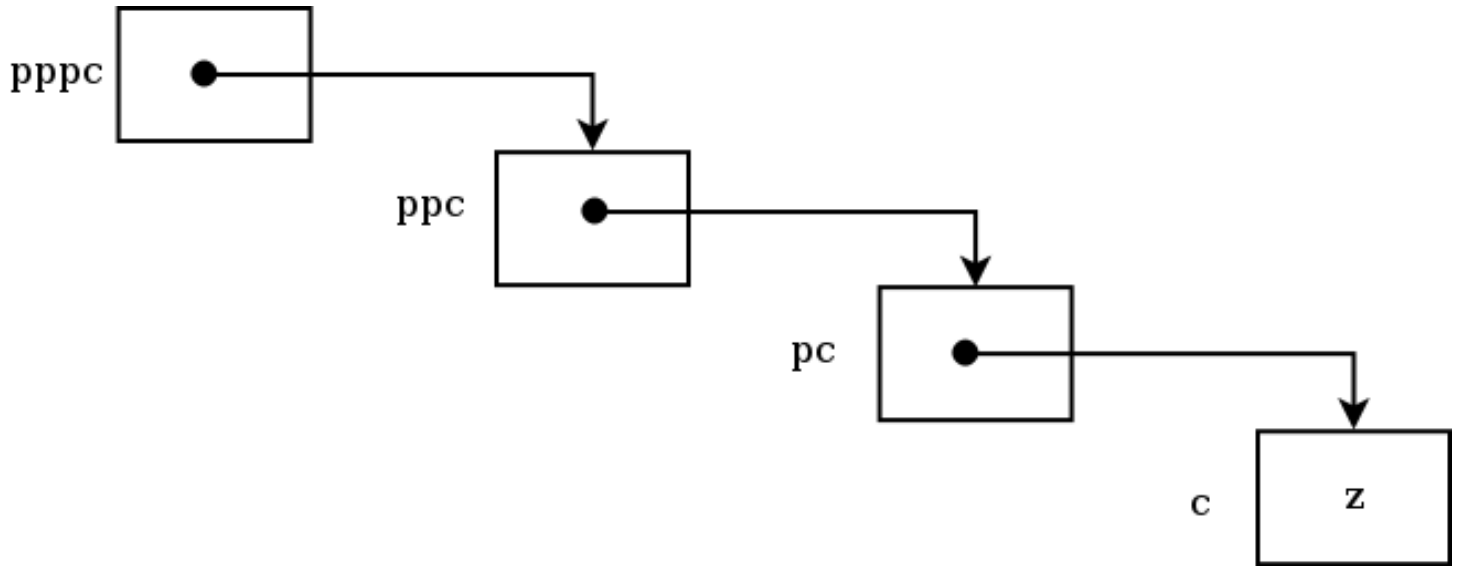
```
char c = 'z';
char *pc = &c;
char **ppc = &pc;
char *** pppc = &ppc;
***pppc = 'm'; // cambia el valor de c a 'm'
```



Punteros

Puntero a puntero

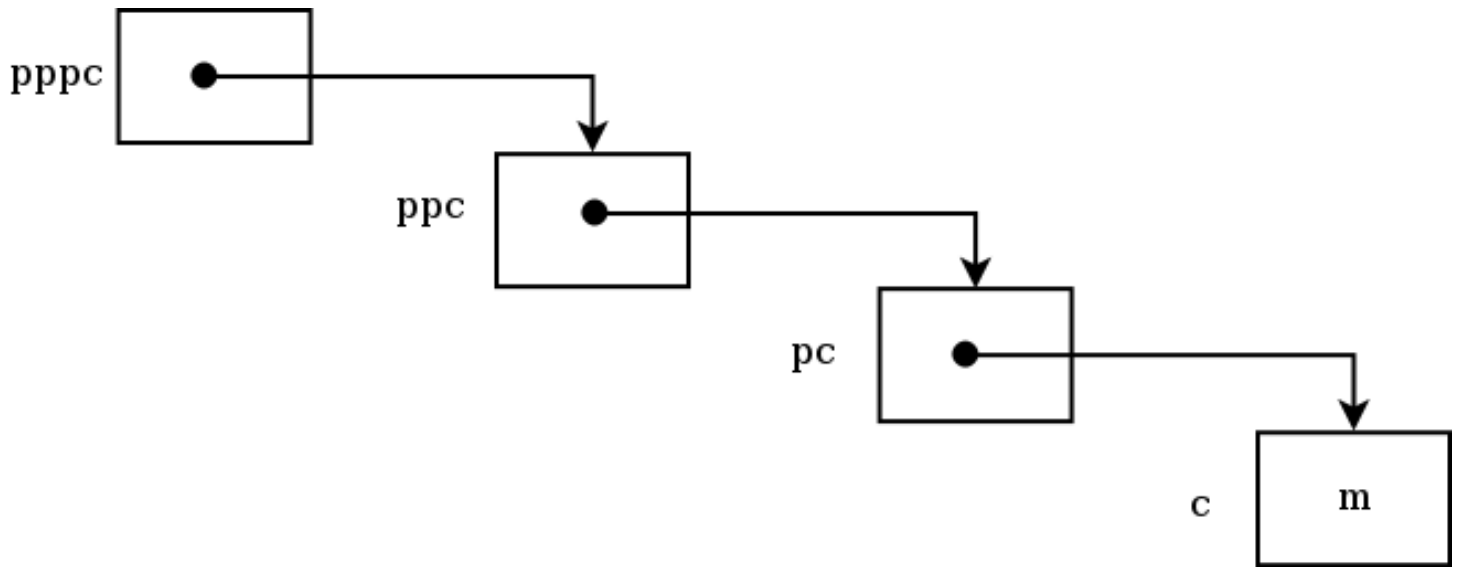
```
char c = 'z';
char *pc = &c;
char **ppc = &pc;
char *** pppc = &ppc;
***pppc = 'm'; // cambia el valor de c a 'm'
```



Punteros

Puntero a puntero

```
char c = 'z';
char *pc = &c;
char **ppc = &pc;
char *** pppc = &ppc;
***pppc = 'm'; // cambia el valor de c a 'm'
```



Punteros

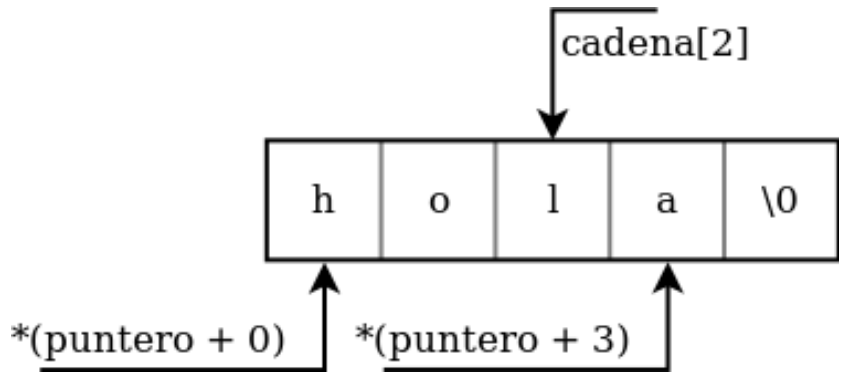
Aritmética de punteros

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  char cadena[5];
  char *puntero;

  puntero = &cadena[0]; /* puntero apunta a cadena[0] */
  *puntero = 'h';       /* cadena[0] = 'h' */
  *(puntero+1) = 'o';   /* cadena[1] = 'o' */
  *(puntero+2) = 'l';   /* cadena[2] = 'l' */
  *(puntero+3) = 'a';   /* cadena[3] = 'a' */
  *(puntero+4) = '\0';  /* cadena[4] = '\0' */

  return 0;
}
```



PREGUNTAS:

1) ¿CÓMO MUESTRO EL VALOR DE LA PRIMERA POSICIÓN DEL ARREGLO USANDO EL PUNTERO?

2) ¿CÓMO MUESTRO EL MISMO VALOR USANDO LA VARIABLE DEL ARREGLO?

3) ¿CÓMO MUESTRO EL CONTENIDO COMPLETO DEL ARREGLO USANDO EL PUNTERO?

4) ¿CÓMO MUESTRO EL CONTENIDO COMPLETO DEL ARREGLO USANDO LA VARIABLE DEL ARREGLO?

Punteros

Operaciones permitidas

Sean `ptr1`, `ptr2` punteros a objetos del mismo tipo, y n un tipo entero o una enumeración; las operaciones permitidas y los resultados obtenidos con ellas son:

Operación	Resultado	Comentario
pt1++	puntero	Desplazamiento ascendente de 1 elemento
pt1- -	puntero	Desplazamiento descendente de 1 elemento
pt1 + n	puntero	Desplazamiento ascendente n elementos [4]
pt1 - n	puntero	Desplazamiento descendente n elementos [4]
pt1 - pt2	entero	Distancia entre elementos
pt1 == NULL	booleano	Siempre se puede comprobar la igualdad o desigualdad con NULL
pt1 != NULL	booleano	
pt1 = pt2	puntero	Asignación
pt1 = void	puntero genérico	Asignación

Punteros

Paso por referencia frente a paso por dirección

```c
#include <stdio.h>

void funcion1(int *i)
{
  *i = *i + 5;
}

void funcion2(int i)
{
  i = i + 5;
}

int main (int argc, char const* argv[])
{
  int i;
  i = 10;
  funcion1(&i); // paso por dirección
  printf("%d\n", i);
  funcion2(i); // paso por referencia
  printf("%d\n", i);
  return 0;
}
```

Salida:

```
15
15
```

Arreglos Dinámicos

- Para definir durante la ejecución del programa, arreglos cuyo tamaño es exactamente el que el usuario necesita.

- Se Utilizan para ello dos funciones de la biblioteca estándar (`stdlib.h`):
  - `malloc()` : Abreviatura de *memory allocate*, que podemos traducir por “reservar memoria”. Ésta solicita un bloque de memoria del tamaño que se indique (en *bytes*).
  - `free()` : Que en inglés significa “liberar”. libera memoria obtenida con malloc, es decir, la marca como disponible para futuras llamadas a `malloc`.

Arreglos Dinámicos

Ejemplo

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6    int *a;
7    int n, i;
8
9    printf ("Número de elementos: ");
10   scanf ("%d", &n);
11   a = malloc(n * sizeof (int));
12   for(i=0; i<n; i++)
13     a[i] = i;
14   free(a);
15   return 0;
16 }
```

Arreglos Dinámicos

Explicación

- `int *a;`
- No se trata de un puntero a un entero, sino de un puntero a una secuencia de enteros.
- Ambos conceptos son equivalentes en C, pues ambos son meras direcciones de memoria.
- La variable `a` es un arreglo dinámico de enteros, pues su memoria se obtiene dinámicamente.
- Esto es, en tiempo de ejecución y según convenga a las necesidades.
- No sabemos aún cuántos enteros serán apuntados por `a`, ya que el tamaño no se conocerá hasta que se ejecute el programa y se lea por teclado.

Arreglos Dinámicos

Explicación

- La función `malloc` espera recibir como argumento un número entero: el número de *bytes* que queremos reservar.
- Si deseamos reservar `n` valores de tipo `int`, hemos de solicitar memoria para `n * sizeof (int)` *bytes*.
- Recordemos que `sizeof(int)` es la ocupación en *bytes* de un tipo de dato (en caso de `int` sabemos que es de 4).

Arreglos Dinámicos

Explicación

- La función `free` recibe un puntero a cualquier tipo de datos: la dirección de memoria en la que empieza un bloque previamente obtenido con una llamada a `malloc`.
- Lo que hace `free` es liberar ese bloque de memoria, es decir, considerar que pasa a estar disponible para otras posibles llamadas a `malloc`.
- Es como cerrar un archivo: si no necesito un recurso, lo libero para que otros lo puedan aprovechar.
- Podemos así aprovechar la memoria de forma óptima.
- **Importante: el programa debe efectuar una llamada a `free` por cada llamada a `malloc`.**

- Si vas a seguir ocupando el puntero, conviene que después de hacer `free` asignes al puntero el valor `NULL`.

- `free` libera la memoria apuntada por un puntero, pero no modifica el valor de la variable que se le pasa.

- Imaginemos que un bloque de memoria de 10 enteros que empieza en la dirección 1000 es apuntado por una variable `a` de tipo `int *`, es decir, imagina que vale 1000.

- Cuando ejecutamos `free(a)`, ese bloque se libera y pasa a estar disponible para eventuales llamadas a `malloc`.

Arreglos Dinámicos

`Free`

- Pero ¡`a` sigue valiendo 1000! ¿Por qué? Porque a se ha pasado a `free` **por valor, no por referencia**, así que `free` no tiene forma de modificar el valor de `a`.
- Es recomendable que asignes a `a` el valor `NULL` después de una llamada a `free`, eso hace explícito que la variable `a` no apunta a nada.
- **Recuerden, es responsabilidad de uno y conviene hacerlo: asignar explícitamente el valor `NULL` a todo puntero que no apunte a memoria reservada.**

- La función `malloc` puede fallar por diferentes motivos.
- Podemos saber cuándo ha fallado nos devuelve el valor NULL.
- Imaginemos que solicitas 2 *megabytes* de memoria en un computador que sólo dispone de 1 *megabyte*.
- En tal caso, la función `malloc` devolverá el valor `NULL` para indicar que no pudo efectuar la reserva de memoria solicitada.

Arreglos Dinámicos

malloc

- Los programas correctamente escritos deben comprobar si se pudo obtener la memoria solicitada y, en caso contrario, tratar el error.

```
1 a = malloc(n * sizeof (int));
2 if(a == NULL)
3   printf ("Error: no hay memoria suficiente\n");
4 else {
5   ...
6 }
```

- Es posible solicitar la memoria y comprobar si se pudo obtener en una única línea:

```
1 if((a = malloc(talla * sizeof (int))) == NULL)
2   printf ("Error: no hay memoria suficiente\n");
3
4 else {
5   ...
6 }
```

Arreglos Dinámicos

`calloc`

- La función `calloc` es similar a `malloc`, pero presenta un prototipo diferente y hace algo más que reservar memoria: la inicializa a cero.
- Se define:

```
1  void * calloc(int nmemb, int size);
```

- Con `calloc`, puedes pedir memoria para un arreglo de `n` enteros así:

```
1  a = calloc(n, sizeof (int));
```

- El primer parámetro es el número de elementos y el segundo, el número de *bytes* que ocupa cada elemento.
- No hay que multiplicar una cantidad por otra, como en el `malloc`.
- Todos los enteros del arreglo se inicializan a cero.

Arreglos Dinámicos

calloc

- Es como si ejecutáramos este fragmento de código:

```
1 a = malloc(n * sizeof (int));
2 for(i = 0; i < n; i++)
3   a[i] = 0;
```

- ¿Por qué no usar siempre `calloc`, si parece mejor que `malloc`?

- Por eficiencia. En ocasiones no desearías que se pierda tiempo de ejecución inicializando la memoria a cero, ya que uno mismo necesitará inicializarla a otros valores inmediatamente.

- Recuerda que garantizar la mayor eficiencia de los programas es uno de los objetivos del lenguaje de programación C.

Ejercicios

1. Desarrolle una función que reciba un entero n y devuelva un arreglo de largo n con los primeros n números de Fibonacci

$$f_n = \begin{cases} 0 & si \quad n = 0 \\ 1 & si \quad n = 1 \\ f_{n-1} + f_{n-2} & si \quad n > 1 \end{cases}$$

2. Desarrolle una función que reciba un arreglo dinámico y recorra sus elementos verificando la paridad de sus elementos.

Cadenas dinámicas

- Las cadenas son un caso particular de arreglos.
- Podemos usar cadenas de cualquier longitud gracias a la gestión de memoria dinámica.
- El siguiente programa, lee dos cadenas y construye una nueva que resulta de concatenar a éstas.

Cadenas dinámicas

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define CAPACIDAD 80

int main(int argc, char *argv[])
{
  char cadena1[CAPACIDAD+1], cadena2[CAPACIDAD+1];
  char *cadena3;
  printf ("Dame un texto: ");
  gets(cadena1);
  printf ("Dame otro texto: ");
  gets(cadena2);
  cadena3 = malloc((strlen(cadena1) + strlen(cadena2) + 1) * sizeof (char));
  strcpy(cadena3, cadena1);
  strcat(cadena3, cadena2);
  printf ("Resultado de concatenar ambos: %s\n", cadena3);
  free(cadena3);
  return 0;
}
```

Ojo: La función `gets()` en la revisión más reciente de la standar C (2011) ha eliminado definitivamente esta función desde su especificación.

Matrices dinámicas

- Podemos extender la idea de los arreglos dinámicos a matrices dinámicas.
- Pero el asunto se complica notablemente: no podemos gestionar la matriz como una sucesión de elementos contiguos, sino como un “arreglo dinámico de arreglos dinámicos”.
- Analicemos el siguiente código.

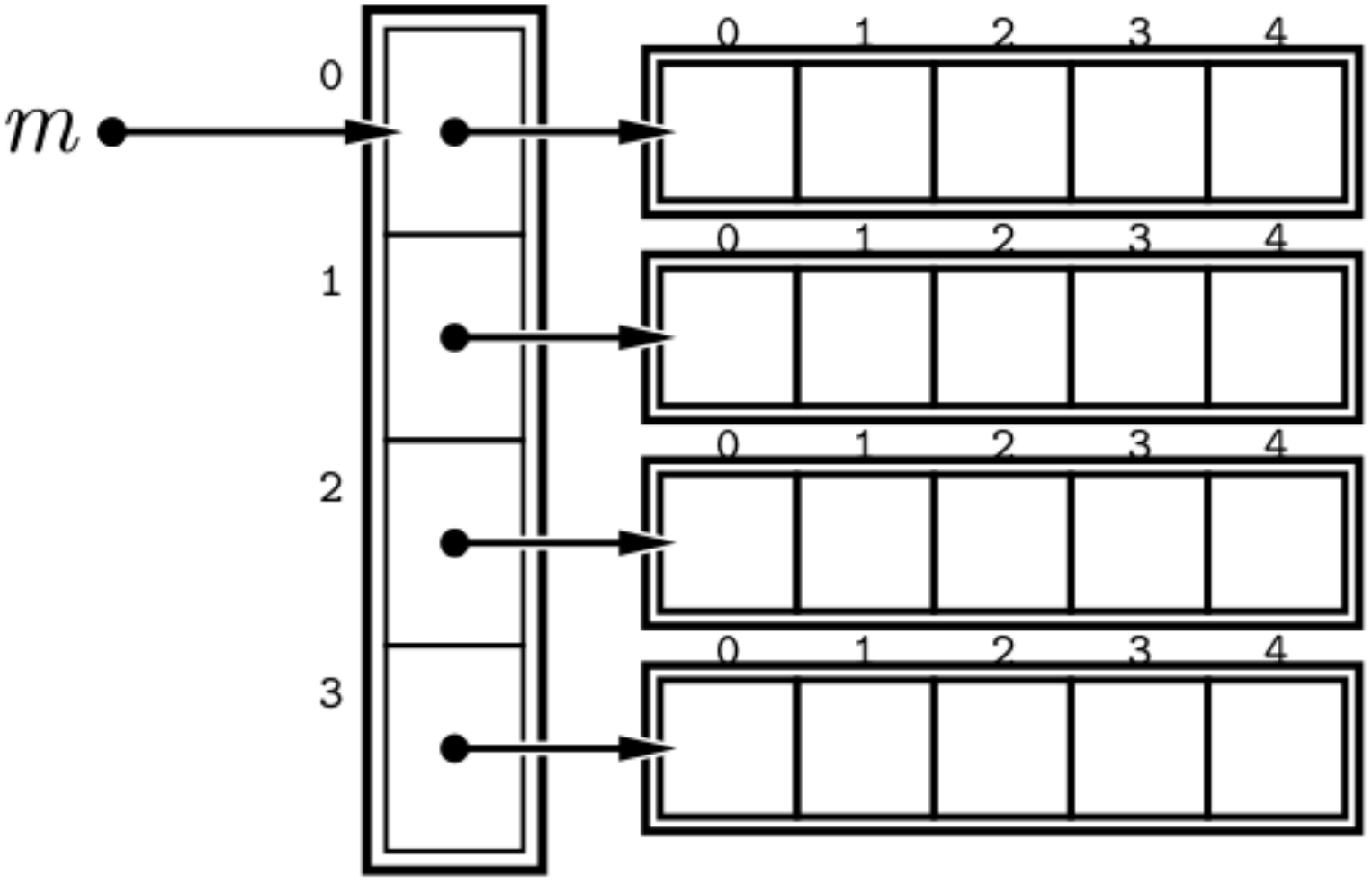
Matrices dinámicas

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  float ** m;
  int i, filas, columnas;

  printf("Filas: ");
  scanf("%d", &filas);
  printf("Columnas: ");
  scanf("%d", &columnas);
  /* reserva de memoria */
  m = malloc(filas * sizeof(float *)) ;
  for(i=0; i<filas; i++)
    m[i] = malloc(columnas * sizeof(float)) ;
  /* trabajo con m[i][j] */
  /* .... */

  /* liberación de memoria */
  for(i=0; i<filas; i++)
    free(m[i]);
  free(m);
  return 0;
}
```

Matrices dinámicas



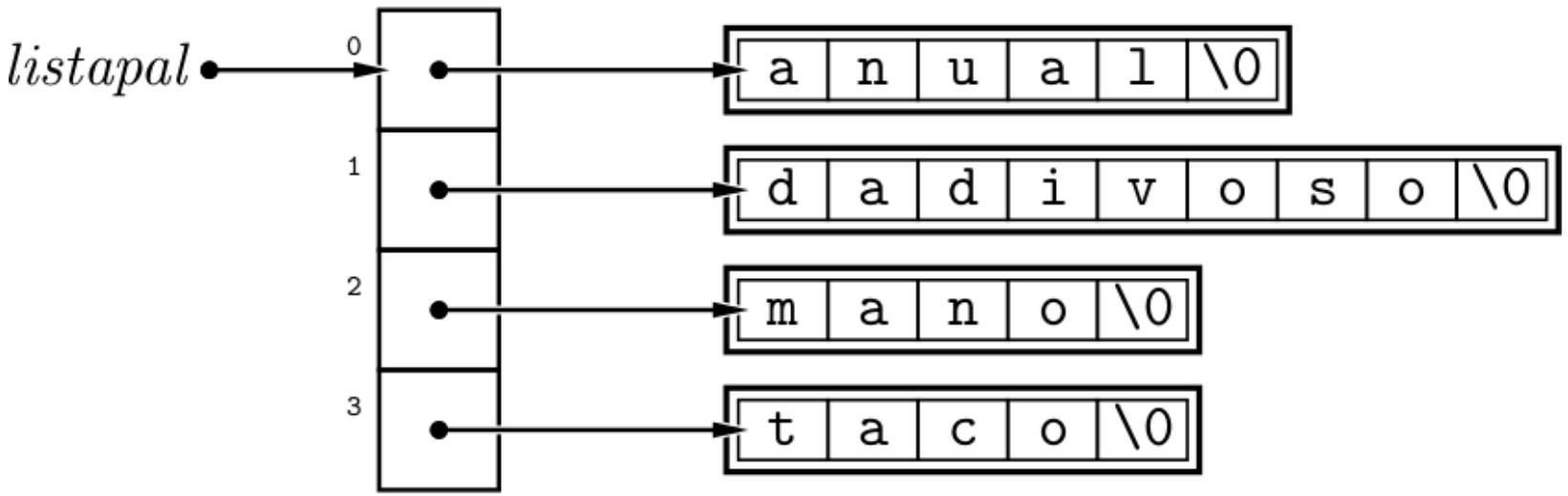
Matrices dinámicas

Arreglos de arreglos de tamaños arbitrarios

- Hemos aprendido a definir matrices dinámicas con un arreglo dinámico de arreglos dinámicos.
- El primero contiene punteros que apuntan a cada columna.
- Una característica de las matrices es que todas las filas tienen el mismo número de elementos (el número de columnas).
- Hay estructuras similares a las matrices pero que no imponen esa restricción.
- Pensemos, por ejemplo, en una lista de palabras.
- Una forma de almacenarla en memoria es como se ve en la siguiente figura.

Matrices dinámicas

Arreglos de arreglos de tamaños arbitrarios



Matrices dinámicas

Arreglos de arreglos de tamaños arbitrarios

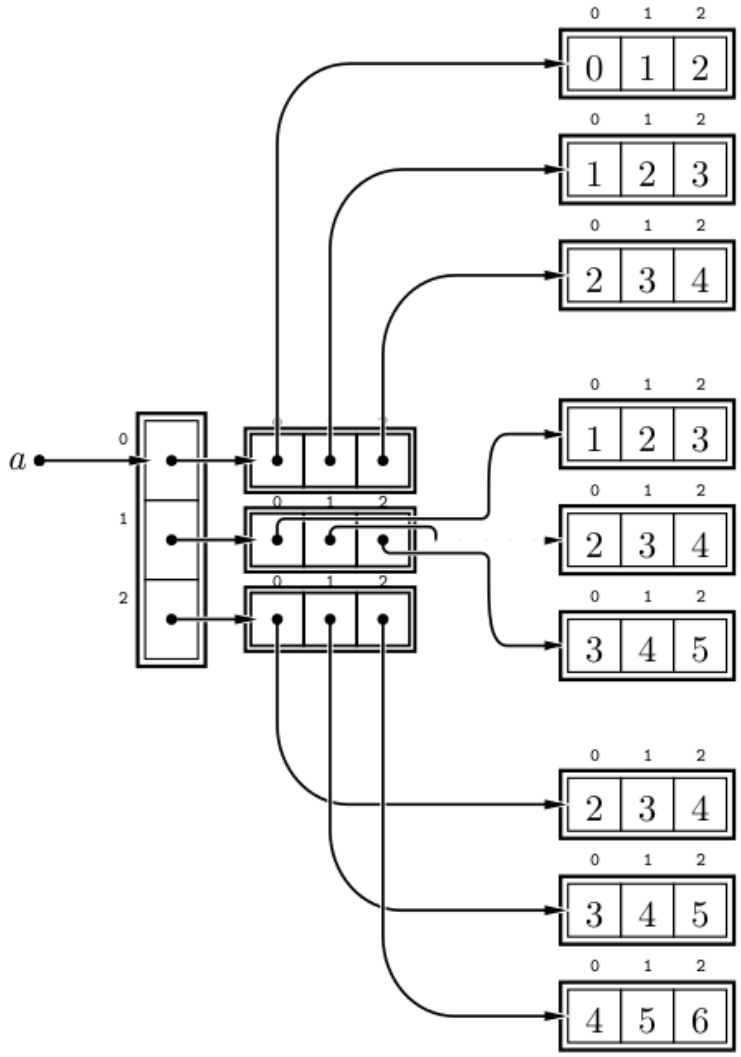
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define PALS 4

int main(int argc, char *argv[])
{
  char **listapal;
  char linea[100];
  int i;
  /* Pedir memoria y leer datos */
  listapal = malloc(PALS * sizeof(char *));
  for(i=0; i<PALS; i++)
    {
      printf ("Teclea una palabra: ");
      scanf("%s", linea);
      listapal [i] = malloc((strlen(linea)+1) * sizeof (char));
      strcpy(listapal [i], linea);
    }
  /* Mostrar el contenido de la lista */
  for(i=0; i<PALS; i++)
    printf ("Palabra %: %s\n", i, listapal [i]);
  /* Liberar memoria */
  for(i=0; i<PALS; i++)
    free(listapal[i]);
  free(listapal);
  return 0;
}
```

Arreglos n-dimensionales

- Hemos considerado la creación de estructuras bidimensionales (matrices o arreglos de arreglos).
- Pero nada impide definir estructuras con más dimensiones.
- Acá un ejemplo sencillo donde se ilustra la idea creando una estructura dinámica con $3 \times 3 \times 3$ elementos, inicializarla, mostrar a su contenido y liberar la memoria ocupada.

Arreglos n-dimensionales



Formar

Transformar

Universidad Andrés Bello

Arreglos n-dimensionales

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  /* Tres asteriscos: arreglo de
     arreglos de arreglos de enteros. */
  int ***a;
  int i, j, k;

  /* Reserva de memoria */
  a = malloc(3*sizeof (int **));
  for (i=0; i<3; i++)
    {
      a[i] = malloc(3*sizeof (int *));
      for (j=0; j<3; j++)
        a[i][j] = malloc(3*sizeof (int));
    }

  /* Inicialización */
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      for (k=0; k<3; k++)
        a[i][j][k] = i+j+k;
```

```c
  /* Impresión */
  for(i=0; i<3; i++)
    for(j=0; j<3; j++)
      for(k=0; k<3; k++)
        printf ("%d %d %d: %d\n",
                i, j, k, a[i][j][k]);

  /* Liberación de memoria. */
  for(i=0; i<3; i++)
    {
      for(j=0; j<3; j++)
        free(a[i][j]);
      free(a[i]);
    }
```

Redimensionar la reserva de memoria

- Muchos programas no pueden determinar el tamaño de sus arreglos antes de empezar a trabajar con ellos.
- Por ejemplo, cuando se inicia la ejecución de un programa que gestiona una agenda telefónica no sabemos cuántas entradas contendrá finalmente.
- Podemos fijar un número máximo de entradas y pedir memoria para ellas con `malloc`.
- Pero entonces estaremos reproduciendo el problema que nos llevó a presentar los arreglos dinámicos.
- Afortunadamente, C permite que el tamaño de un arreglo cuya memoria se ha solicitado previamente con `malloc` crezca en función de las necesidades.
- Se usa para ello la función `realloc`.

Redimensionar la reserva de memoria

```c
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int * a;
  a = malloc(10 * sizeof (int)); /* Se pide espacio para 10 enteros. */
  /* ... */
  a = realloc(a, 20 * sizeof (int)); /* Ahora se amplía para que quepan 20. */
  /* ... */
  a = realloc(a, 5 * sizeof (int)); /* Y ahora se reduce a sólo 5 (los 5 primeros). */
  /* ... */
  free(a);
  return 0;
}
```

Estructuras con punteros

- En este caso, la estructura requiere almacenar una cantidad de elementos que se saben en tiempo de ejecución.
- Para ellos usamos un puntero.
- Para el ejemplo, usamos una lista de notas y la cantidad de ellas.
- Para acceder a los elementos de dicha lista, se acceden de forma habitual, con "*", "(", ")" y "." (punto).

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  struct nota{
5      int *notas;
6      int cantidadNotas;
7  };
8
9  typedef struct nota Nota;
```

```
10 int main(){
11     Nota notasAlumno;
12     notasAlumno.cantidadNotas = 3;
13     notasAlumno.notas = (int *)malloc(sizeof(int)*notasAlumno.cantidadNotas);
14     *notasAlumno.notas = 6;
15     *(notasAlumno.notas+1) = 5;
16     notasAlumno.notas[2] = 4;
17     return 0;
18 }
```

Punteros a estructuras

- Se reserva en forma dinámica elementos del tipo de una estructura.
- Para poder acceder a ellos, se realiza de dos formas principalmente:
  - Operador flecha: `->`: Accede directamente a los elementos.
  - Operador "`.`" y `*`: Se utiliza el operador de indirección (`*`) para ingresar a la estructura y después con el . para poder acceder al elemento.
  - Operador "`.`" y `[]`: se maneja como si fuera un arreglo.

Punteros a estructuras

Ejemplo

```
1  struct alumno{
2      char nombre[50];
3      int edad;
4      Nota *nota;
5  };
6  typedef struct alumno Alumno;
7
8  int main(){
9      Alumno *al = (Alumno*)malloc(sizeof(Alumno));
10     al->edad=20;
11     strcpy(al->nombre, "Pedro");
12
13     Alumno *conjuntoAlumnos = (Alumno*)malloc(sizeof(Alumno)*2);
14     conjuntoAlumnos->edad = 21;
15     (conjuntoAlumnos+1)->edad = 22;
16     (*(conjuntoAlumnos+1)).edad = 24;
17     strcpy(conjuntoAlumnos[0].nombre, "Ana");
18     strcpy(conjuntoAlumnos->nombre, "Pablo");
19     return 0;
20 }
```

Puntero a estructura a estructura

- Es posible acceder a los elementos de una estructura que sea apuntado por un puntero.
- Se utilizan en combinación de las formas anteriores.

```
1  conjuntoAlumnos->nota = &notasAlumno;
2  conjuntoAlumnos[1].nota = (Nota*) malloc(sizeof(Nota));
3
4  conjuntoAlumnos[1].nota->notas = (int*)malloc(sizeof(int));
```

Puntero a estructura a estructura

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct _nota_{
  int *notas;
  int cantidadNotas;
};
typedef struct _nota_ Nota;

struct alumno{
  char nombre[50];
  int edad;
  Nota *nota;
};
typedef struct alumno Alumno;
```

```c
int main(){
  Nota notasAlumno;
  notasAlumno.cantidadNotas = 3;
  notasAlumno.notas = (int *) malloc(sizeof(int)*
      notasAlumno.cantidadNotas);

  *notasAlumno.notas = 6;
  *(notasAlumno.notas+1) = 5;
  notasAlumno.notas[2] = 4;

  Alumno *al = (Alumno*)malloc(sizeof(Alumno));
  al->edad=20;
  strcpy(al->nombre, "Pedro");

  Alumno *conjuntoAlumnos = (Alumno*)malloc(sizeof(
      Alumno)*2);

  conjuntoAlumnos->edad = 21;
  (conjuntoAlumnos+1)->edad = 22;
  (*(conjuntoAlumnos+1)).edad = 24;
  strcpy(conjuntoAlumnos[0].nombre, "Ana");
  strcpy(conjuntoAlumnos->nombre, "Pablo");
  conjuntoAlumnos->nota = &notasAlumno;
  conjuntoAlumnos[1].nota = (Nota*) malloc(sizeof(Nota))
      ;

  conjuntoAlumnos[1].nota->notas = (int*)malloc(sizeof(
      int));

  *((*((*(conjuntoAlumnos+1)).nota)).notas)= 7;
  return 0;
}
```

Argumentos de la línea de comandos

- Los programas que diseñamos en el curso generalmente en el `main` tienen argumento.

```
int main(int argc, char *argv[])
```

- La función `main` recibe como argumentos las opciones que se indican en la línea de comandos cuando ejecutas el programa desde la consola de GNU/Linux.
- `$ ./saluda -n nombre`

Argumentos de la línea de comandos

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  if (argc != 3)
    printf ("Error: necesito que indiques el nombre con -n\n");
  else
    if (strcmp(argv [1], "-n") != 0)
      printf ("Error: sólo entiendo la opción -n\n");
    else
      printf ("Hola, %s.\n", argv [2]);
  return 0;
}
```